Algoritmos y Estructura de Datos: Examen 1 (Solución)

Grados Ing. Inf. y Mat. Inf. Octubre 2015

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Importante:

- **Cada pregunta debe responderse en hojas diferentes**.
- Todas las hojas entregadas deben indicar **apellidos**, **nombre** y **DNI/NIE**.
- Este examen dura **90 minutos** y consta de **3 preguntas** que puntúan hasta **10 puntos**.
- Las calificaciones provisionales de este examen se publicarán el **5 de noviembre de 2015** en el Moodle de la asignatura el junto con el procedimiento para solicitar la revisión.

(3½ puntos) 1. **Se pide:** Implementar en Java el método:

```
public static PositionList<Integer> flatNub (PositionList<Integer> [] arr)
```

que toma como parámetro un «array» `arr` que no será null y cuyos elementos no serán null sino que referenciarán objetos listas de posiciones cuyos elementos tampoco serán null sino que referenciarán objetos de tipo `Integer`. Si `arr` es vacío entonces el método debe devolver una nueva lista vacía. En otro caso, el método debe recorrer el «array» de izquierda a derecha y por cada lista del mismo recorrer ésta de izquierda a derecha insertando al final de la lista resultado los elementos que no estén ya en la lista resultado. El interfaz `PositionList<E>` está disponible en el apéndice de este examen.

Por ejemplo, sea `arr` un «array» de tamaño 3 donde `arr[0]` referencia la lista [0,2,5,1], `arr[1]` referencia la lista [1,3,7,2] y `arr[2]` referencia la lista [4,1,2,5]. La lista resultado debe ser [0,2,5,1,3,7,4]. Puede asumirse que se dispone de la clase `NodePositionList<E>` que implementa listas de posiciones, y que también se dispone del método auxiliar

```
private static <E> boolean member(E elem, PositionList<E> list)
```

que indica si un elemento `elem` aparece en la lista `list`.

Solución:

```
public static PositionList<Integer> flatNub (PositionList<Integer> [] arr) {
  PositionList<Integer> res = new NodePositionList<Integer>();
  if (arr.length == 0) return res;  // podría quitarse esta línea
  for (int i = 0; i < arr.length; i++) {
    Position<Integer> cursor = arr[i].first();
    while (cursor != null) {
      if (!member(cursor.element(),res))
        res.addLast(cursor.element());
      cursor = arr[i].next(cursor);
    }
  }
  return res;
}
```

(3½ puntos) 2. **Se pide:** Implementar en Java el método

```
public static boolean iguales (Integer [] arr1, Integer [] arr2)
```

que toma como parámetro dos «arrays» `arr1` y `arr2` de objetos de clase `Integer` e indica si los «arrays» son de la misma longitud y tienen los mismos elementos de izquierda a derecha. Se asume que `arr1` y `arr2` no son null pero pueden contener elementos null.

Solución:

```
public static boolean iguales (Integer [] arr1, Integer [] arr2) {
  if (arr1 == arr2 || arr1.length == 0 && arr2.length == 0) return true;
  if (arr1.length != arr2.length) return false;
  int i;
  for (i = 0; i < arr1.length && !eqElem(arr1[i],arr2[i]); i++)
    ;
  return i == arr1.length;
}
private static boolean eqElem (Integer i1, Integer i2) {
  return i1 == null && i2 == null || i1 != null && i1.equals(i2);
}
```

(3 puntos) 3. Se tiene el siguiente interfaz `ColorRGB`:

```
public interface ColorRGB {
 public int getRed();    // Devuelve el nivel de rojo del color
 public int getGreen();  // Devuelve el nivel de verde del color
 public int getBlue();   // Devuelve el nivel de azul del color
 public String getHex(); // Devuelve la representación hexadecimal del color
}
```

Se tiene también el código incompleto de las clases `ColorHex` y `ColorInts` que implementan dicho interfaz.

```
public class ColorHex implements ColorRGB {
  private String hex;

  // Los constructores y métodos auxiliares ya están implementados...

  public int getRed() { /* COMPLETAR */ }
  public int getGreen() { /* COMPLETAR */ }
  public int getBlue() { /* COMPLETAR */ }
  public String getHex() { /* COMPLETAR */ }
  public boolean equals(Object o) { /* COMPLETAR */ }
}

public class ColorInts implements ColorRGB {
  private int r, g, b;

  // Los constructores y métodos auxiliares ya están implementados...

  public int getRed() { /* COMPLETAR */ }
  public int getGreen() { /* COMPLETAR */ }
  public int getBlue() { /* COMPLETAR */ }
  public String getHex() { /* COMPLETAR */ }
  public boolean equals(Object o) { /* COMPLETAR */ }
```

La clase `ColorHex` utiliza un atributo de tipo `String` para representar el color como un valor haxedecimal. La clase `ColorInts` utiliza tres atributos entero para representar los niveles de rojo, verde y azul del color. Asumimos que ya están implementados todos los constructores y los siguientes métodos auxiliares:

```
private int getRedFromHex(String hex);  // obtiene el rojo de un hexadec.
private int getBlueFromHex(String hex); // obtiene el azul de un hexadec.
private int getGreenFromHex(String hex);// obtiene el verde de un hexadec.
private String getHexValue(int r, int g, int b); // rgb a hexadec.
```

Los tres primeros permiten obtener de un hexadecimal el nivel de rojo, azul y verde. El último permite obtener el valor hexadecimal a partir de los niveles de rojo, azul y verde.

Se pide: Completar los «getters» y el método `equals` de cada clase. El método `equals` debe implementarse de forma que un objeto de clase `ColorHex` (idem `ColorInts`) indique si es igual o no a cualquier objeto de cualquier clase que implemente el interfaz `ColorRGB`. Es decir, el método `equals` debe indicar si dos colores son iguales independientemente de la representación.

```
public class ColorHex implements ColorRGB {
  private String hex;
  ...
  public int getRed()    { return getRedFromHex(this.hex);   }
  public int getGreen()  { return getGreenFromHex(this.hex); }
  public int getBlue()   { return getBlueFromHex(this.hex); }
  public String getHex() { return this.hex; }
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o instanceof ColorRGB) {
       ColorRGB c = (ColorRGB) o;
       return this.getHex().equals(c.getHex());
    } else
      return false;
  }
}
public class ColorInts implements ColorRGB {
  private int r, g, b;
  ...
  public int getRed()    { return this.r; }
  public int getGreen()  { return this.g; }
  public int getBlue()   { return this.b; }
  public String getHex() { return getHexValue(this.r,this.g,this.b); }
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o instanceof ColorRGB) {
       ColorRGB c = (ColorRGB) o;
       return this.getHex().equals(c.getHex());
    } else
      return false;
  }
}
```

Interfaz `Position<E>`

```
package positionList;
/** Interface for abstract nodes. */
public interface Position<E> {
  public E element();
}
```

Interfaz `PositionList<E>`

```
public interface PositionList<E> extends Iterable<E> {
  /** Returns the number of nodes in the list. */
  public int size();
  /** Returns true if the list is empty (has no nodes), false if it's not. */
  public boolean isEmpty();
  /** Returns the first node of the list, or null if the list is empty. */
  public Position<E> first();
  /** Returns the last node of the list, or null if the list is empty. */
  public Position<E> last();
  /** Either returns the next node to the right of parameter node 'p' in the
   *  list, or returns null if 'p' doesn't have a next node to the right.
   *  Raises the exception if 'p' is null or is not a node of the list.
   */
  public Position<E> next(Position<E> p) throws IllegalArgumentException;
  /** Either returns the previous node to the left of parameter node 'p' in
   *  the list, or returns null if 'p' doesn't have a previous node to the
   *  left. Raises the exception if 'p' is null or is not a node of the list.
   */
  public Position<E> prev(Position<E> p) throws IllegalArgumentException;
  /** Inserts a new first node to the list with 'elem' as element. */
  public void addFirst(E elem);
  /** Inserts a new last node to the list with 'elem' as element.  */
  public void addLast(E elem);
  /** Inserts a new node with 'elem' as element to the list right before
   *  parameter node 'p'. Raises the exception if 'p' is null or is not a node
   *  of the list.
   */
  public void addBefore(Position<E> p, E elem) throws IllegalArgumentException;
  /** Inserts a new node with 'elem' as element to the list right after
   *  parameter node 'p'. Raises the exception if 'p' is null or is not a node
   *  of the list.
   */
  public void addAfter(Position<E> p, E elem) throws IllegalArgumentException;
  /** Removes node 'p' from the list and returns its element. Raises the
   *  exception if 'p' is null or is not a node of the list.
   */
  public E remove(Position<E> p) throws IllegalArgumentException;
  /** Sets the element of node 'p' on the list to 'elem' and returns the old
   *  element in 'p'. Raises the exception if 'p' is null or is not a node of
   *  the list.
   */
  public E set(Position<E> p, E elem) throws IllegalArgumentException;
  /** Returns an array with all the elements of the list or an empty array
   *  of length zero if the list is empty.
   */
  public Object [] toArray();
}
```